Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Fault-Tolerant Control System Modeling in SystemC

Tiago Pereira Vidigal

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Daniel Chaves Café

Brasília
2017

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Fault-Tolerant Control System Modeling in SystemC

Tiago Pereira Vidigal

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Daniel Chaves Café (Orientador)
EnE/UnB

Prof. Dr. Gilmar Silva Beserra
UnB/FGA

Prof. Dr. Henrique Cezar Ferreira
EnE/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
Coordenador do Curso de Engenharia da Computação

Brasília, 06 de Janeiro de 2017

Dedicatória

Essa tese é dedicada a

minha mãe Lilian, por me mostrar que o estudo é o investimento mais precioso,
minha noiva Denise, por me dar coragem, auxilio e amor além de tudo,
meus amigos mais próximos, pelas risadas e ajuda nos momentos mais importantes.
Amo todos vocês.

Agradecimentos

Eu gostaria de agradecer Prof. José Edil Guimarães de Medeiros, Prof. Daniel Chaves Café, Prof. Daniel Mauricio Muñoz Arboleda e Prof. Gilmar Silva Beserra por toda a ajuda ao longo da execução deste trabalho. Vocês me deram a direção para eu chegar onde estou agora. Conto com seus conselhos de agora em diante. Eu gostaria de agradecer também a DFchip, onde eu pude aprender muito sobre microeletrônica digital. Com trabalho duro e investimento, eu encontrei minha verdadeira área de interesse e espero trabalhar nela enquanto eu puder.

Resumo

Este trabalho tem como objetivo a criação de uma metodologia de modelagem de sistemas de controle tolerantes a falta, de uma forma que confiabilidade possa ser medida e tolerancia a falta possa ser validada. A necessidade de controladores confiáveis requer o uso de técnicas de segurança. A validação dessas técnicas é importante e deve ocorrer cedo no fluxo de projeto. Modelos são usados nesses cenários para maiores níveis de abstração. No entanto, modelos devem ter técnicas de injeção de faltas para possibilitar testes com faltas. Uma metodologia usando um Modelo de Computação (MOC) com SystemC é proposto juntamente com um *framework* para injeção de falta. Um caso de estudo valida a metodologia e o *framework*, dando a possibilidade de avaliar a confiabilidade de um modelo.

Palavras-chave: tolerancia a faltas, controle, injecao de faltas, systemc, modelo

Abstract

This work has the objective to create a methodology of modeling a fault-tolerant control system, in a way that reliability can be measured and fault tolerance can be validated. The need of reliable controllers requires the use of dependability techniques. It's important to validate those techniques as early as possible in the project flow. Models are used in this scenarios for higher levels of abstraction. However, models must have a fault injection technique to enable tests with faults. A methodology using an heterogeneous Model of Computation (MOC) with SystemC is proposed together with a fault injection framework. A case of study validates the methodology and the framework, giving the possibility to evaluate reliability of a model.

Keywords: fault tolerance, control system, fault injection, systemc, model

Contents

List of Figures

Chapter 1

Introduction

The text blends many knowledge areas. This introduction chapter gives a brief explanation of this topics and presents ideas to the reader that will be further detailed in-depth. An overview of safety-critical systems, control systems, models and faults introduces the general idea of this work.

1.1 Context

Safety-critical systems deals with situations where failure can lead to life loss and environment disasters, like avionic systems for example [1]. The economic success of avionic systems depends on the safety provided by the aircraft, turning fault tolerance essential in that context [2]. Designers must bear in mind that physical components may fail during regular operation or in error-prone and unpredictable conditions. That leads to designs that are tolerant to a finite number of faults avoiding an eventual overall system failure or complete breakdown.

Fault tolerance is necessary because it is practically impossible to build a perfect system. It is the property that enables a system to continue operating properly in the event of a failure. A fundamental compromise of safety-critical systems is complexity versus reliability. The increase in the number of features, sensors and embedded electronics makes complexity rise. Since faults are likely to be caused by situations outside the control of the designers, the reliability of the system drastically decreases unless compensatory measures are taken [1].

Control systems are present in various systems. The development of dependable controllers can increase reliability of these systems. The increase of complexity of systems require more abstract levels for explorations and tests at earlier stages. The reliability of a system must be verified as early as possible to guarantee low probability of failure during operation.

In avionic systems, controllers are present to operate control surfaces, which are movable parts located in the wings and tail to execute movements while the plane is in the air. Faulty surfaces may cause failure and result in an accident. Aircrafts have several types of air movements (yaw, roll and pitch), executed by control surfaces (elevators, flaps, ailerons and rudders) [3]. Those parts must be precisely controlled for the correct operation of the aircraft. The design of control surface controllers must be carefully planned to achieve the desired resilience.

Even though simulators of complex systems at system-level can be handy tools, they come with a caveat. Simulators only run what they are told to and usually, as modelers, we don't consider faulty situations. We are used to model ideal scenarios. Faults must be created with dedicated fault injection techniques that simulate faulty components.

Safety-critical systems have an important role in our society. The correct modeling of them and constant validation are essential for a robust and fault-tolerant system. A methodology to guide designers interested in developing those types of systems increases production efficiency and reduces possible wrong design choices.

1.2 Goals

The main goal of this work is to propose a methodology of modeling a fault-tolerant control system, in a way that reliability can be measured and fault tolerance can be validated. A secondary goal is to implement a case of study to test the methodology. It's also intended to develop a fault injection framework for fault tolerance validation.

This chapter gives the context and the goals of the work. Chapter 2 details all the concepts used to achieve the goal. That way, the reader can understand terms used and choices made by the author. Chapter 3 describes not only the methodology proposed to model fault-tolerant control systems, but also suggestions of testbench architecture and fault tolerance techniques. A fault injection framework is proposed by the author. Chapter 4 uses the methodology to develop a control system to manipulate an aircraft's elevator. The proposed fault injection framework is used to validate the reliability of the fault tolerance techniques. Finally, chapter 5 gives the conclusion of this work and proposes future works derived from it.

Chapter 2

Theories about Modeling and Fault

The methodology to create a model of a fault-tolerant control system requires a set of concepts. The focus of this chapter is to give an overview of them. The main ideas used in this work are of control systems, processes and faults, each of them detailed in a section.

Section 2.1 gives all the theory of control systems used in this work. Classical composition and details about controllers are given. State-space representation is briefly explained to model plants.

Section 2.2 gives a process used to develop electronic circuits projects. The process is detailed and the importance of modeling is highlighted. Formal definitions and classifications are presented, as well as an overview of the SystemC language.

Section 2.3 explains fault related information. Fault classifications and tolerance techniques are listed, giving a broad idea of how to improve dependability. Fault injection techniques are also listed, focusing in implementations in SystemC.

2.1 Control Systems

A control system uses a specified stimulus to obtain a desired response. They are classified in two major configurations: open loop and closed loop. Open loop configuration has the disadvantage of being sensitive to disturbances and unable to correct them. The closed configuration compensate disturbances, has a greater accuracy and are less sensitive to noise and environment changes. However, it may be unstable [4].

Control system with a closed loop configuration typically have a controller that drives a plant to reach an setpoint. The plant is the process driven by the controller, receiving the control action or gain. It receives the controller's gain and generates the system's output. This output is desired to be equal to setpoint, so their difference is used to compute the output error (feedback loop). Then, the controller computes a new gain based on setpoint and current error values [4].

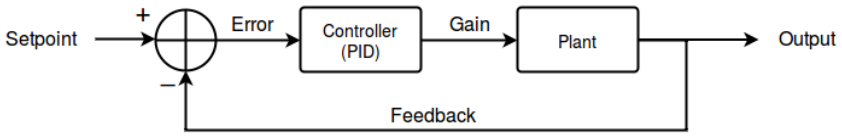

Figure 2.1: Classic closed-loop configuration control system.

Closed loop control systems can have two configurations of compensation: cascade or feedback. The cascade configuration uses an compensator in series with the original controller and the feedback simply returns the plant's output to a subtractor, which computes the difference with the setpoint and send it to the compensator. The feedback configuration is similar, but the compensator is inserted at the feedback loop, parallel with the original controller. Feedback can yield faster responses, but the cascade compensation is simpler [4].

The Proportional-Integral-Derivative (PID) controller is a traditional compensator in cascade configuration. It's composed by an active Proportional-Derivative (PD) controller followed by an Proportional-Integral (PI) controller. PID uses three coefficients for it's computation: *Kp*, *Ki* and *Kd*. They are respectively related to the proportional, integral and derivative terms [4].

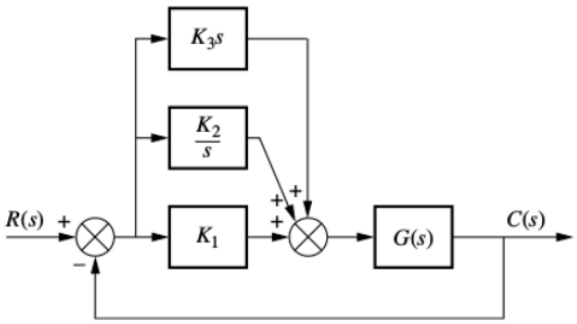

Figure 2.2: PID controller in block diagram at S-Domain. K1, K2 and K3 are Kp, Ki and Kd, respectively (Source: [4]).

The plant is driven by the controller and can be modeled in the frequency or time domains. The state-space is a mathematical model of a system in the time domain, represented by the equations 2.1. x is state vector and $\dot{x}$ is the derivation in respect to time of it. y is output value and u is input or control vector. A, B, C and D are respectively system, input, output and feedfoward matrices [4].

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned} \tag{2.1}$$

2.2 Electronic System Project Flow

Electronic systems are present in our daily routine. They increased in presence and complexity along the years. Integrated circuits containing high density of components give the possibility to have small and low power consuming circuits. Some of these type of circuits are Application Specific Integrated Circuit (ASIC) and System-on-a-Chip (SoC). However, the complexity of the design process of those circuits is challenging.

The system design can be a daunting process for SoC designs. The modeling of multiple architectures and their refinement is required to achieve an appropriate structure. The process for the design involves several steps: create the system specification, develop a behavioural model, refine and test it, determine decomposition, specify and develop a hardware architectural model, refine and test it with cosimulation, specify implementation blocks [5].

The system specification determines the system requirements, which inform required functions, performance, cost and development time. The behavioural model is a high-level model that simulates the final system's functionality. It creates a executable specification, a software implementing the whole system behaviour, which is used to validate specifications, test algorithms and possibly used as reference model. The decomposition divides the modeled systems in two partitions to be developed: software (programs run by a processor) and hardware (circuits). The hardware architecture provides exploration and determines blocks to be developed, communication between them, memories to be used and others. It provides a detailed specification of the functionality, performance and interfaces of the system and its blocks [5].

The modeling at various steps of the design process represented at figure 2.3 shows the importance of this technique. Models help to abstract the details of complex systems to simplify the design and planning of them. However, systems are getting continuously more complex and higher levels of abstraction are required. Languages must be modified or created to attend that demand of system-level modeling.

Model of Computation (MOC) is the term used to determine a group of properties and features to characterize a type of model. A definition proposed by Lee and Sangiovanni-Vincetelli [6] uses a mathematical language to give a formalism to terms and enable description, abstraction and differentiation of models of computation. The framework is a "meta model" and don't define completely MOCs, but permits the comparison of notions of concurrency, communication and time between them [6].

MOCs are described using tagged signals, systems and composition of systems. Tags are used to model time, precedence relationship, synchronization points and other key properties of MOC. Signals are set of events, which have a tag and a value. A process is a set of possible behaviours, signals that satisfies a process. Tag systems are sets of

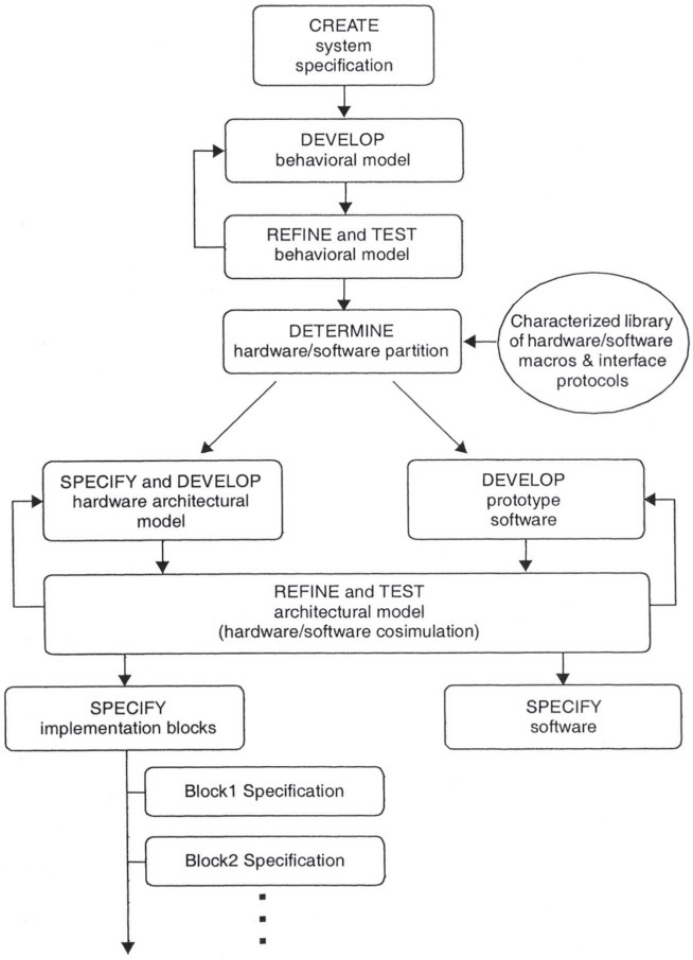

Figure 2.3: Design process of a SoC (Source: [5]).

processes that can be timed, untimed or heterogeneous. Timed MOCs have a tag system with a totally ordered set of tags, which are also called time stamps. Untimed MOCs have partially ordered tags and their ordering generally denotes causality or synchronization. Heterogeneous models mix both tag systems [6].

Traditional design languages generally have a single fixed MOC with little or no customization [7]. SystemC is a versatile language that enable system-level modeling that emulates a large number of MOCs [7]. It was created by the Open SystemC Initiative (OSCI), which joined Accellera Organization and The SPIRIT consortium to create Accellera Systems Initiative. SystemC language turned an IEEE standard (IEEE 1666) in 2005 and it's extension for TLM was merged in 2011. Accellera defines SystemC as a language build in standard C++ by extending it with a set of class libraries and used worldwide [8].

SystemC's architecture is layered. Base layers provide event driven simulation, data

types and base components. On top of this foundation, MOCs and methodologies specific layers are added for flexible system modeling. The base layers permit to create the basic structure. Modules represent the building blocks of the system. The communication between them uses ports, interfaces and channels. Processes are functions inside modules and can be configured as threads or methods. Events are used to trigger processes [7].

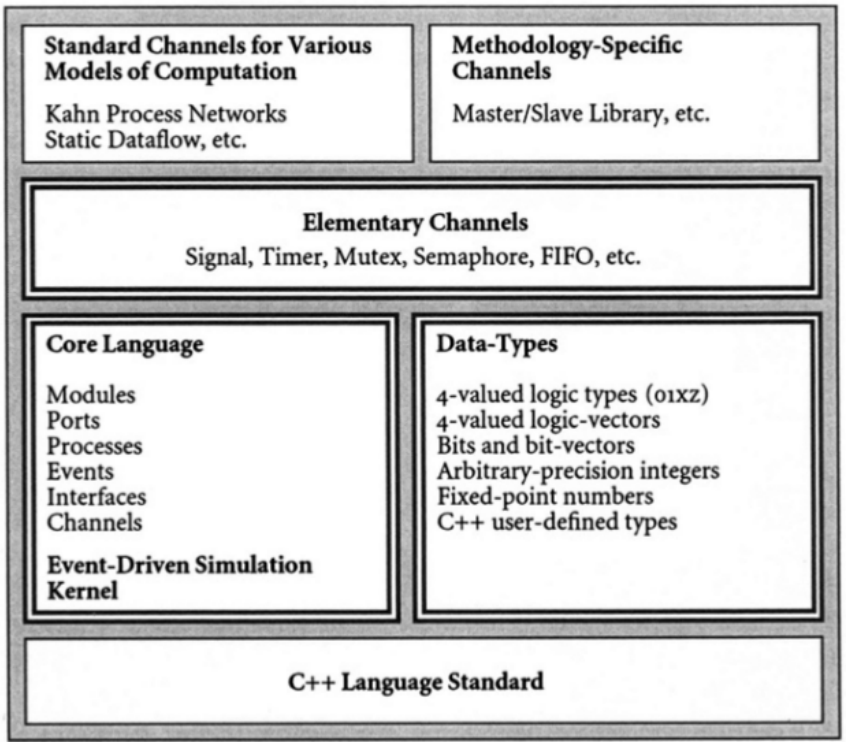

Figure 2.4: Layers of the SystemC's architecture (Source: [7]).

The layered architecture enables the multiple MOC types implementation. Not only models in Register-Transfer Level (RTL), the language also enables creation of Transport Level Model (TLM). Using definitions of Lee and Sangiovanni-Vincetelli [6], the language can model timed models, like discrete event and discrete-time, and untimed models, like Kahn Process Networks (KPN) and Static Dataflow (SDF) [7].

TLM designs, compared with RTL, is a higher abstraction level approach. It's better for explorations of architecture and implementation as you can, for example, change the communication channel with ease. It's relatively easy to develop, understand and extend. It also provides faster simulations [7].

In the TLM extension of SystemC, initiator and target ports permit to send and receive transports of data, respectively. Transports are generated creating an generic payload and sending it with an initiator port. A target port triggers an event when a transport is received. The event triggered executes a predefined process inside the module with the

target port. The top level module connects initiator and target ports, so communication is established [7].

The design of an application may be represented with diagrams. The common notation for TLM is big squares to represent modules, small squares for initiators, small circles for targets and arrows for connection. However, languages to develop this kind of model generally support object-oriented programming. A more complete approach is required to create meaningful diagrams. The Unified Modeling Language (UML) is an interesting standard developed by the Object Management Group (OMG). It defines a common structure for structural modeling. The abstract syntax define notations to create diagrams of programs with the object-oriented paradigm. A full description can be found at [9].

2.3 Faults of a System

Safety-critical systems require to be reliable. Reliability is the ability of the system to deliver its service without interruption until a determined point of time. However, the complexity of systems continually increases, which causes its reliability to drastically decrease. Compensatory measures must be taken to increase dependability, the ability of a system to deliver its intended level of service to its user [1].

The interruption of the service delivery if a system is called **failure**. This occurs because of **errors**, difference between actual and expected results. **Fault** is a physical defect, imperfection or flaw that generates error. Faults can be classified by it's source and duration [1].

Fault sources are separated in four groups: incorrect specification, incorrect implementation, fabrication defects and external factors. The first three groups are sources of defects, while external factors are sources of transient faults. Hardware faults are classified into permanent, transient and intermittent faults according to it's duration. Permanent faults remain active until they are corrected. Transient faults remain active for a short period of time. Intermittent faults are transient ones that are active periodically [1].

Considering that failures are generated by faults, the reliability of a system can be increased using dependability means. These are methods and techniques to develop dependable systems, such as fault tolerance, prevention, removal and forecast. Fault tolerance is the ability of a system prevent to fail even with occurrence of faults by using redundancy. Fault prevention attempts to prevent introduction or occurrence of faults. Fault removal target the reduction of the number of faults present in the systems. Fault forecast aims to estimate present and future faults, as well as their consequences [1].

Fault tolerance and Fault forecast focus at hardware faults from fabrication defects and external factors. Fault prevention has a broader focus, using design reviews for incorrect

specification and implementation as well as shielding and security means for external factors. Fault removal focus at incorrect implementation and maintenance during life time [1].

These techniques have the objective to increase the dependability, which have some common measures and concepts. The failure rate is defined as the expected number of failures per unit time and the chance of a system failing increases over time. The reliability of a system with constant failure rate decreases exponentially, following the equation 2.2 with constant λ, with is called exponential failure law [1].

$$R(t) = e^{-\lambda t} \tag{2.2}$$

Hardware redundancy can improve dependability and is an interesting approach in situations in which equipment cannot be maintained or there's a high cost for doing it. Redundancy is additional implementation unnecessary at fault-free environments but, otherwise, helps to maintain the functionality of the system. It can be present as space or time redundancy. Space redundancy relies on adding components. Time redundancy uses the same components to recompute result and compare with previous one [1].

Hardware redundancy is achieved with physical copies of components. It has three types: passive, active and hybrid. Passive hardware redundancy simply filters faults without requiring intervention. Active hardware redundancy removes faulty components of the system. Hybrid hardware redundancy filters faults and replaces faulty component [1].

Hybrid hardware redundancy combines advantages of both passive and active approaches. They're generally used in safety-critical applications once it prevents momentary erroneous results and enables reconfiguration of the system to correct it. Two possible techniques are self-purging and N-modular redundancies. Self-purging performs an constant voting between the multiple identical parallel components and purges the ones which result differs from the others. N-modular redundancy is similar to self-purging, but purged components are substituted by spare ones [1].

Fault tolerance generates more complexity for the already complex systems developed nowadays. The modeling of such complexity in higher abstraction levels is desirable. Software models operate in ideal fault-free environments, so the reliability of the system can't be directly inferred. Fault injection is the common approach to validate implementation and verify robustness of fault-tolerant systems [10]. Fault injection is the action of inserting faults in a fault-free simulation.

Shafik, Rosinger and Al-Hashimi [11] list some classical fault injection techniques: saboteurs, mutants, simulation command. Saboteurs are fault injection components placed between two modules, altering value or timing of a signal. Mutants are modified

components that replace fault-free modules and inject faults. Simulation command-based injection focus in variable and signal manipulation in run time.

The proposed fault injection technique in the work of Shafik, Rosinger and Al-Hashimi [11] have an interesting modular centralized structure. Fault injection enabled types are used to replace primitive C++ and SystemC types of the fault-free model and are automatically registered in a database class during build phase. A fault injection policy class controls the location and probability of faults and the manager class injects the faults using the same clock as the system.

Chang and Chen [10] suggest the use of a distributed approach in contrast with Shafik, Rosinger and Al-Hashimi's propose. The communication channels are modified to insert fault with fault injection modules and other components. These frameworks where created for various levels of abstraction: bus-cycle accurate, untimed functional transaction and timed functional transaction. This methodology requires significant modification of the system.

Chapter 3

Methodology

The many steps of modeling in the design flow presented in section 2.2 make evident the importance of model creation. In the case of safety-critical systems, models include fault tolerance techniques. These require a fault injection framework to validate them and verify the resulting reliability of the system.

Section 3.1 proposes a control system architecture and explains the PID algorithm form. A discrete form of the state-space equations of section 2.1 enables the implementation of them in a programing language. A testbench architecture is proposed to validate the model using Matlab and Simulink.

Section 3.2 explains more some fault tolerance techniques using redundancy. The resulting reliabilities of the system using them are generated, so the final model's reliabilities can be validated. The fault injection framework proposed by the author is explained based on techniques presented in section 2.3.

3.1 Control System Model

A classical closed-loop control system is composed by a subtractor, a controller and a plant. Controllers actually manipulate actuators, which drive a plant. This actuators can be motors for example, probably influencing in the effective control action of the plant.

The connection between control system's modules can use various types of communication methods. This modules may even be distributed along the system that contains the control circuit. This specification makes the use of a TLM to model this type of system an interesting approach. Anyway, the mixture of types of models to create heterogeneous models may be good in some situations. Considering controllers self-contained structures, the familiar RTL can be used to model them, requiring a wrapper to adapt the communication. Figure 3.1 illustrates this propose using a PID controller.

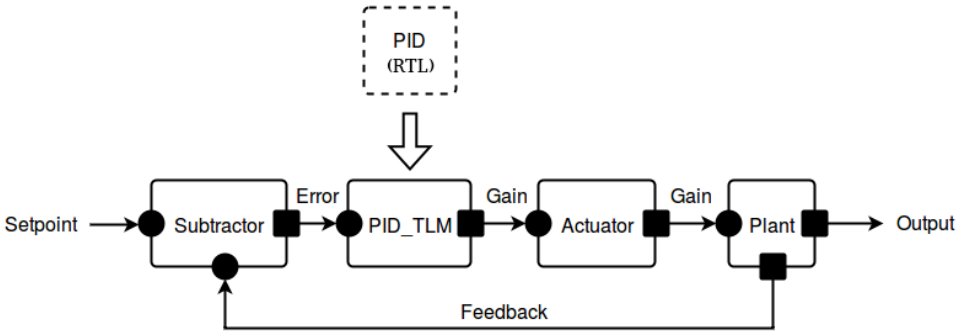

Figure 3.1: Diagram of the control system in TLM with PID controller in RTL.

The SystemC, as discussed in section 2.2, can implement multiple types of MOCs. Besides, it's open source object-oriented language and also enables to merge commonly used MOCs, like RTL and TLM. These are strong points that makes SystemC a recommended language to model systems.

A good programming practice is to separate declaration from definition, where generally header and source files are used, respectively. An alternate style recommended by Black et al.[12] follows this separation idea. Header files (.hpp) contain declaration of fields and methods, as well as documentation annotations. The documentation used follows standards of Doxygen, a tool for generating documentation from annotated C++ sources [13].

Systems must use fault tolerance techniques to increase dependability. Controllers often are electronic systems, so those techniques generally are applied to them. Once exploration is an important point of modeling, the possibility to easily implement and simulate multiple types of fault-tolerant controllers is encouraged. In object-oriented paradigm, an abstract class to represent a generic controller may be used as the base of many child classes, each of them implementing a different technique. In a more specific scenario, an abstract PID controller forces the creation of other PID controllers.

The PID controller has the error of the system as input and the control action as output. It's transform function can be describe as an algorithm and modeled in a programming language [14]. The output gain is calculated as the sum of the proportional, integral and derivative terms. The proportional term is proportional coefficient Kp times the actual error. The integral term is integral coefficient Ki times the sum of actual and past errors. The derivative term is derivative coefficient Kd times the difference of actual and past errors.

Plants are generally represented with transfer functions, with the control action δ as input. The state-space representation briefly explained in section 2.1 is recommended because it's in the time domain, so it's discretization in iterations K can be computed.

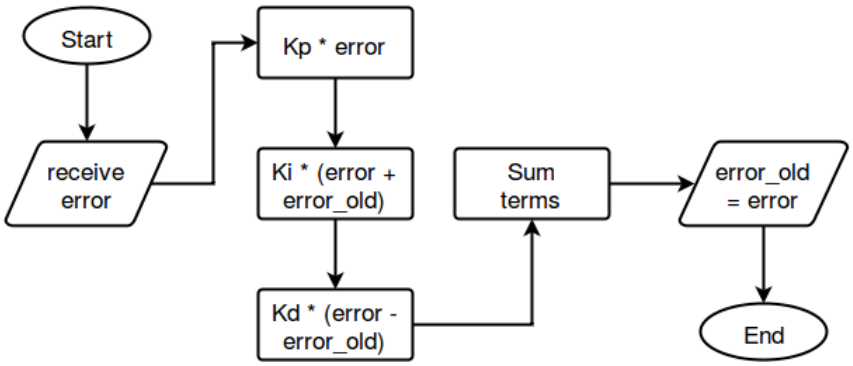

Figure 3.2: Flowchart of PID calculation.

The plant compute the output of the system and also the next state variable values, which are given by the sum of the actual values with their variations. The continuous variation of the state variables $\dot{x}$ of the equations 2.1 must be discretized for computation. An integration method must be used to compute Δx, discrete variation of the state variables, resulting in equation 3.1.

$$\begin{cases} \dot{x} = A \times x(K) + B \times \delta(K) \\ y(K) = C \times x(K) + D \times \delta(K) \\ \Delta x(K) = integration\_method(\dot{x}) \\ x(K+1) = x(K) + \Delta x(K) \end{cases} \tag{3.1}$$

Matlab is a language that control systems are easily modeled with. It can be used to develop a reference model to validate the modules and compute maximum absolute error. A testbench using a black-box test with a Simulink creates input and expected data. A Matlab script stores these values in two separate text files. For simulation equivalence with your model, Simulink simulation's step can be configure with fixed length. Integration method must also be the same as the one you implement.

Stimulus class in your model read the input data and send to the Design Under Test (DUT). Monitor class receive the output of the DUT, read the expected data and compare them. The program stores the received values in a results text file: result.txt. A mean.txt file shows the maximum absolute error between the values. Finally, a second Matlab script read the expected and results text files. A plot shows the comparison of the values for further analysis. The structure presented, illustrated in figure 3.3, can be used for all the modules of the model.

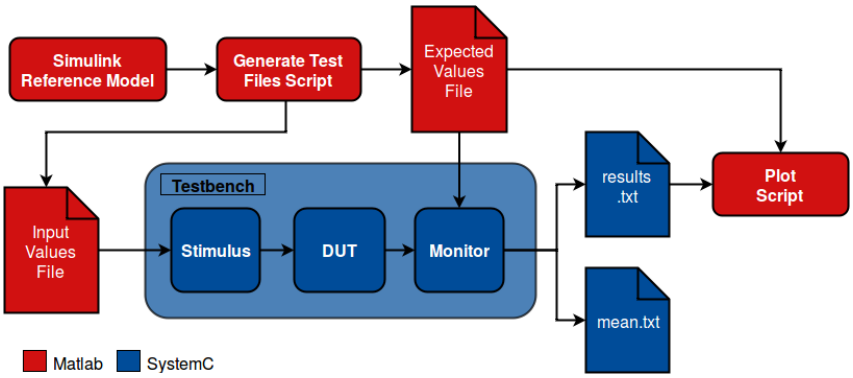

Figure 3.3: Testbench structure used for black-box tests using SystemC.

3.2 Fault Tolerance and Injection

Safety-critical systems must contain a high dependability and faults from all sources must be avoided. Incorrect specification and implementation are originated before the project's transaction phase. However, faults from fabrication defects and external factors occur after product's delivery and are out of designer's control. It's likely to occur faults at this scenario [1].

Fault tolerance techniques using redundancy permits the system to don't fail even with the occurrence of faults from fabrication defects or external factors. The Triple Modular Redundancy (TMR) is an passive hardware redundancy, but can be used together with an self-purging approach. Three PID controllers in parallel are constantly compared with a voter component. If a faulty controller generates a different gain compared with the other two, it is purged. The first of the remaining internal PIDs that fails will make the PID in TMR configuration fail. The figure 3.4 compares the reliability of a system with and without TMR considering the exponential failure law explained at section 2.3.

The TMR/Simplex redundancy is similar to the TMR with self-purging configuration. A faulty internal PID which gain is different of the other two is purged, however one of the correct Simplexes is purged too. This approach uses the benefit of TMR while no component fails and get around the scenario with only two PIDs. Two components have a lower time to fail compared to a single one.

A fault analysis in Matlab compares the reliabilities of the three techniques: TMR/Simplex, TMR and Simplex (without fault tolerance techniques). Mathematical models of this techniques are used to generate the time-to-fail of multiple iterations. Three faults occur in random times following an exponential distribution. Simplex fails with the first fault. TMR fails with the second earliest fault. TMR/Simplex fails with an arbitrary fault different from the earliest. Figure 3.5 shows the comparison of each implementation.

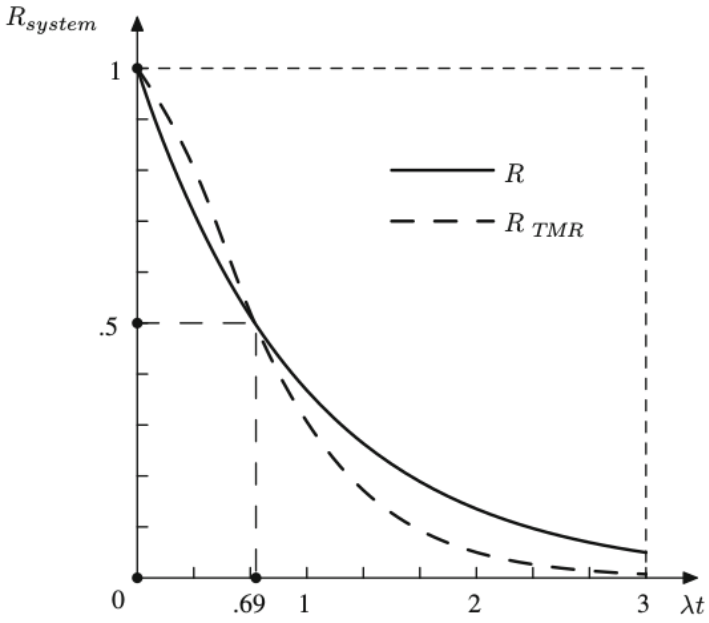

Figure 3.4: TMR reliability as a function normalized time λt (Source: [1]).

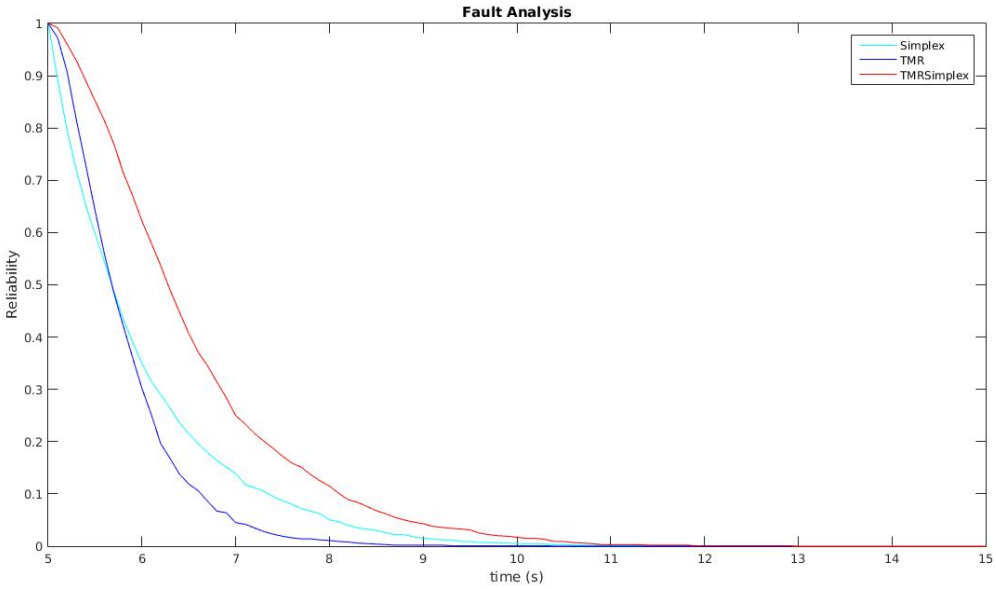

Figure 3.5: Expected reliabilities of each redundancy implementation.

A fault injection framework is necessary to validate the fault tolerance techniques implemented, as well as to infer the reliability of the system with the multiple PIDs developed. A modular implementation requiring minimum modification of the working code is wanted.

The classical approaches require relevant modifications in the system. Saboteurs require to redo the connections to insert the fault modules. Mutants require to edit the code of modules to add fault injection capabilities. Simulation command-based don't require such code or connection modifications.

Shafik, Rosinger and Al-Hashimi's [11] simulation command-based propose has an interesting centralized structure and very little modification of the system modeled. However, implementation details were not fully comprehended. Chang and Chen's [10] saboteur propose has a simpler implementation focused in the communication ports, but the modification required of the system and the distributed fault injection modules where not interesting for this work.

A possible fault injection framework is suggested by using a mixture of both implementations mentioned above. The modular centralized structure of Shafik, Rosinger and Al-Hashimi with the focus in the communication ports for multiple levels of abstraction of Chang and Chen presents a intuitive and versatile way to inject faults.

Communication ports are substituted by their faulty version in the system to enable fault injection. This faulty version is derived from the original port that can hold a fault when it's injected. During build phase, faults set by the module that contains the faulty port are registered in an unique fault injection database together with their time to inject, moment in time when it's going to be injected. During simulation phase, a fault injection manager will check for pending faults in the database and inject them.

Chapter 4

Case of Study: Pitch of an Aircraft

A case of study was implemented to validate the methodology proposed. A system modeling of an aircraft's pitch is available at [15]. This website gives the transfer function and space-state representation of the model. This example was used to develop a validated Simulink control system using PID controller, which was the reference model of this work.

Aircrafts are safety-critical systems, as pointed at chapter 1. Fault tolerance is important in those cases, so fault-tolerant techniques must be inserted in the system to increase reliability. The validation of those requires a fault injection framework to simulate fault occurrence during simulation. The implementation of the complete system model in SystemC is available online: `github.com/TPVidigal/faulty_modeling`.

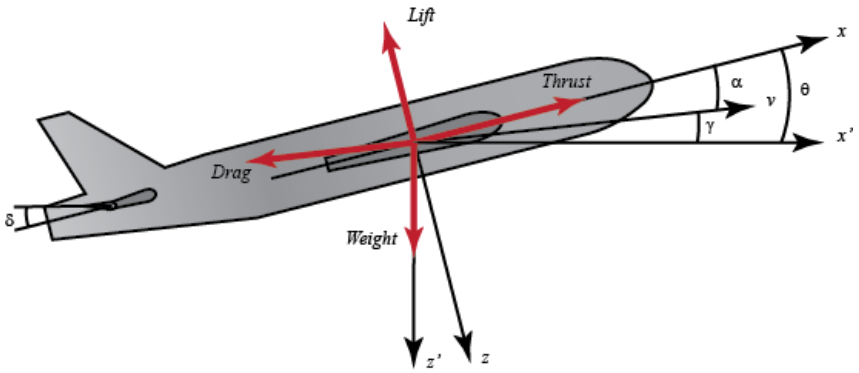

Figure 4.1: Aircraft pitch's dynamics (Source: [15]).

The pitch is the movement that makes the aircraft rise or lower its nose and its dynamics are explained in [15]. The pitch angle θ is defined as the angle between the reference horizontal axis x' and reference line of the body x, relative to the plane's position. The output change is defined by the angle δ of the control surface elevator. The pitch rate q defines the instantaneous angular velocity of θ. The angle of attack α is defined by the flight path v, which angle γ is relative to x'.

The aircraft's pitch control system has the objective to maintain θ with a desired value θ'. The pitch angle is then defined as the **output** of the system and θ' as the **input**. Their difference is defined as **error**. As explained, the angle δ changes the value of θ, so is defined as the **control action**. It is computed with the error of the system by a PID **controller**. The control surface moves with a motor, which is the **actuator**. The motion dynamics described before is defined as the **plant** and its state-space model parameters are described at equation 4.1, using the same values as in [15].

$$
\begin{aligned}
x &= \begin{bmatrix} \alpha \\ q \\ \theta \end{bmatrix} \\
u &= \delta \\
A &= \begin{bmatrix} -0.313 & 56.7 & 0.0 \\ -0.0139 & -0.426 & 0.0 \\ 0.0 & 56.7 & 0.0 \end{bmatrix} \\
B &= \begin{bmatrix} 0.232 \\ 0.0203 \\ 0.0 \end{bmatrix} \\
C &= \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \\
D &= 0
\end{aligned}
\tag{4.1}
$$

4.1 Control System Implementation

The control system modeled contain all the modules presented in figure **??**. This modularized approach increases maintainability, so it's simpler to adapt the model for other cases of study. Iterations of the simulation will happen in fixed steps of 0.1 seconds and a step function is used as input of the complete system, which is always zero before 5 seconds and equals 0.2 from 5 seconds until the end of the simulation. The value 0.2 indicates the aircraft output's angle θ in radians.

The PID controller abstract class in SystemC required a pure virtual property. The property chose was the method doPID, which implements the control action value logic. The input and output ports were instantiated from class sc_buffer instead of sc_in and sc_out. The SystemC input and output classes only trigger an event with write operations with value change, breaking the intended event cascade. The SystemC buffer class, however, triggers an event for any write operation

The TLM wrapper PID_TLM has a pointer that contains the instance of a PID. Any derived class of PID abstract can be inserted in this wrapper. It waits a transport at

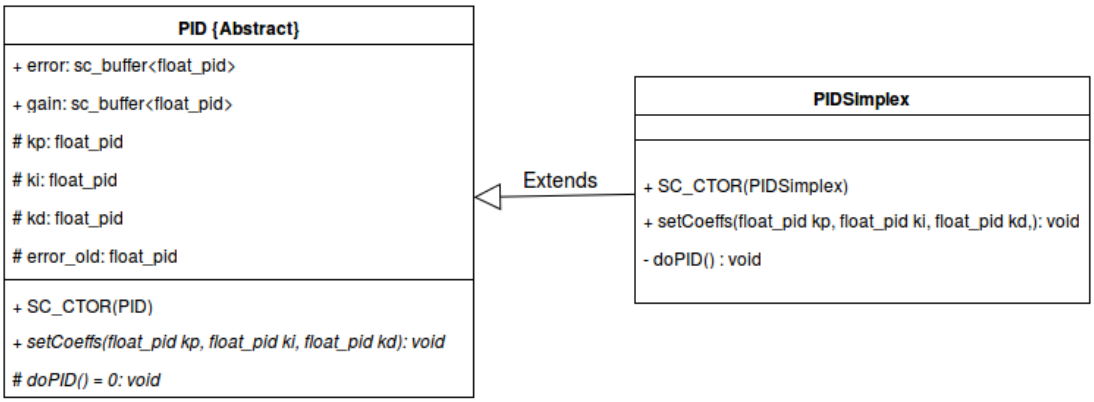

Figure 4.2: UML diagram of PID abstract class and PIDSimplex.

trgtError target port, read the error value stored and writes it at the error buffer of PID. The PID's computed control action value is written at the gain buffer, which triggers the initiator initGain to send the control action.

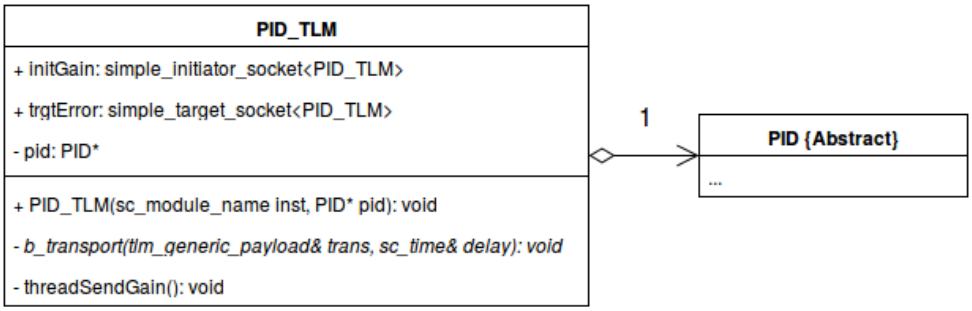

Figure 4.3: UML diagram of PID_TLM wrapper class.

The PID controller in simplex configuration had a straightforward implementation using the algorithm in flowchart at figure 3.2. The same steps were implemented in both SystemC and Matlab languages. The testbench tb_PID_Matlab compares and generates the plot at figure 4.4 from both programs using the testbench architecture defined in section 3.1. The text files were error.txt for input and gain.txt for expected output.

A not expected discontinuity in the Matlab model can be observed at 5.1 seconds of simulation time in figure 4.4. The SystemC model doesn't show this behaviour, causing error between the models. The maximum absolute error presented is of 0.014453. This value is defined as ERROR_PID and is used as a margin of error at the testbench's monitor. In this work, this difference was considered acceptable.

The actuator performs two operations on the control action: saturation and rate limiter. The saturation models the elevator's extreme positions, restricting the value to

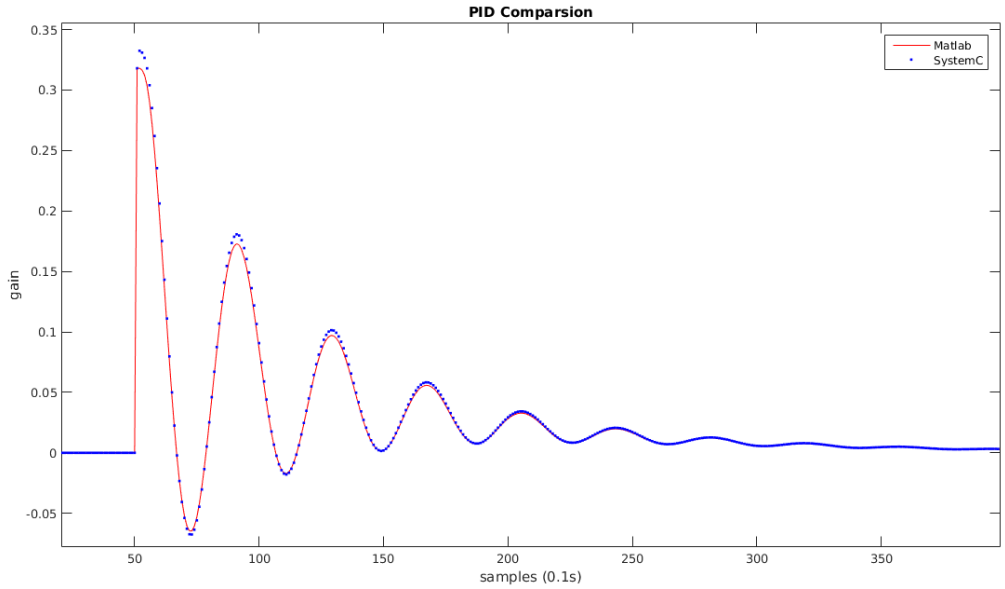

Figure 4.4: Comparison between Matlab and SystemC's implementations of PIDSimplex.

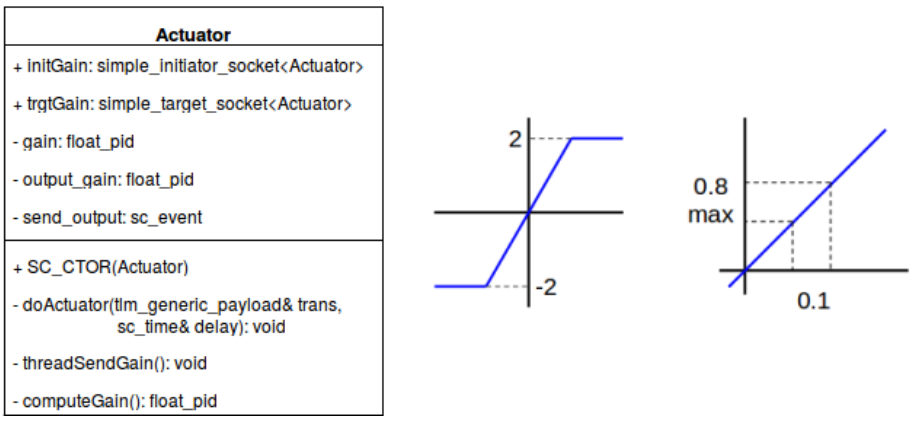

Figure 4.5: UML diagram of Actuator and operations of saturation and rate limiter, respectively.

be between 2 and -2 radians. The rate limiter models the elevator's maximum acceleration, restricting the value's change by 0.8 radians between iterations.

The text files for tb_Actuator were gain.txt for input and output.txt for expected output. The comparison shows a very good match between Matlab and SystemC's implementations. A margin of error ERROR_ACTUATOR is considered for rounding.

The plant resolves iterations with state-space equations. The matrices are named after Nise's notation, detailed at section 3.1. The matrix representation and operations where centralized at the class Matrix. This case of study doesn't contain the matrix

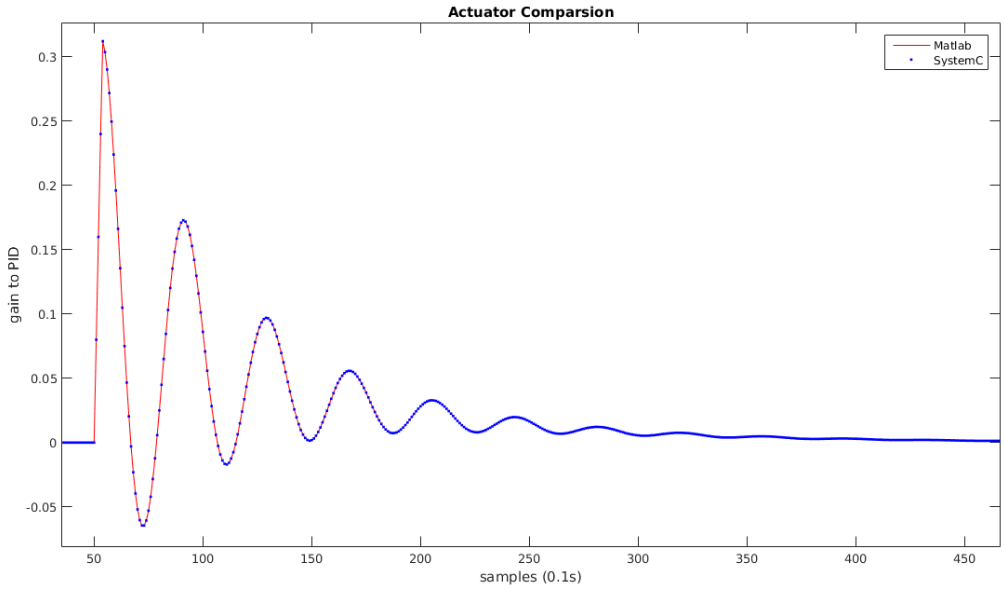

Figure 4.6: Comparison between Matlab and SystemC's implementations of Actuator.

D, so it's omitted for simplicity. Initial values for the matrices m_A, m_B and m_C are defined by the constants MATRIX_A, MATRIX_B and MATRIX_C, respectively. The matrix state is used as the state variables vector of the current iteration. The method PlantFunction implements the iteration, computing output and the next state variables values. The Euler's integration method was chose to compute Δx because of it's simplicity. The difference of time between iterations is fixed and equal to the constant CLK_PERIOD.

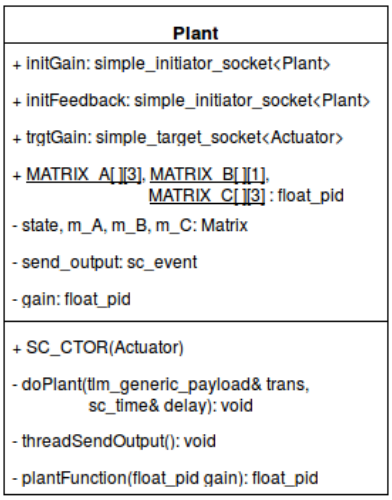

Figure 4.7: UML diagram of Plant.

The text files for tb_Plant were gain.txt for input and output.txt for expected output. The comparison shows a very good match between Matlab and SystemC's implementations. A margin of error ERROR_PLANT is considered for rounding.

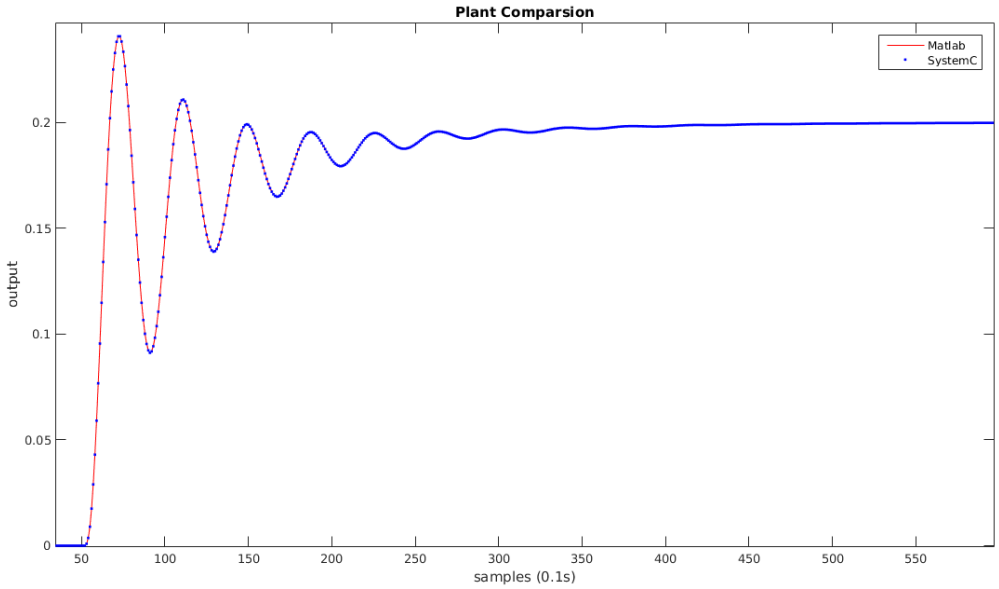

Figure 4.8: Comparison between Matlab and SystemC's implementations of Plant.

The validated blocks were united and the system top level was generated. The text files for tb_System_Matlab were setpoint.txt for input and output.txt for expected output.

The behaviour of the curves is similar, however there is discrepancy between the models. The Matlab implementation present a smaller offset and a faster convergence compared to the SystemC one for the same PID controller coefficients. The maximum absolute error presented is of 0.05. This value is defined as ERROR_SYS and is used as a margin of error at the testbench's monitor.

The difference presented in figure 4.9 may be explained by the Matlab strange PID behaviour indicated in figure 4.4. This hypothesis is reinforced by the lack of error presented in the testbenches of other modules. However, no conclusive result was obtained.

Altogether, the behaviour of the system works as expected, even though the SystemC system is taking more time to reach it's steady state then the Matlab one. The main focus of the case of study is the fault tolerance techniques and injection. Because of that and lack of time, no further tests were performed to resolve those errors.

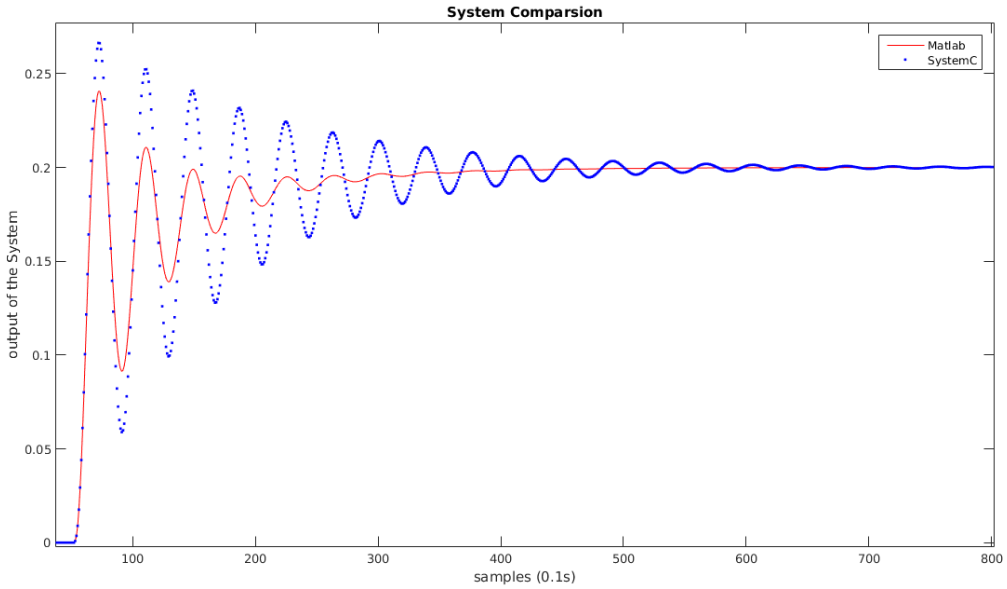

Figure 4.9: Comparison between Matlab and SystemC's implementations of the system.

4.2 Fault Tolerance and Injection Implementation

The model developed in SystemC simulates in a scenario without faults. Fault-tolerant PID controllers in this scenario will work identically to PIDSimplex. Faults are required to validated their tolerance, so a fault injection framework is required.

The two fault tolerance techniques implemented were TMR and TMR/Simplex. Both present the same structure: PIDSimplexes in triple redundancy and a voter. The difference between them are the purging logic, as explained in section 3.2. TMR/Simplex can be considered a modification of the TMR.

The similarity of the chose techniques suggests the creation of a base class PIDTMR derived from PID abstract. The output logic, implemented in the process sendOutputGain, selects a value from a working PID controller as the output. This output logic pretends the purge logic, so derived classes like PIDTMRSimplex must only overload this virtual method.

The approach explained above removed the responsibility of the voter of defining the output of the PID controller. The Voter class is simplified and only determines which internal PID presented error. The PIDSimplex outputs are compared and if one of them is different from the others, an error is detected and the respective flag rises. For example, if the output of the PIDSimplex 1, called gain_1, is different from gain_0 and gain_2, the flag errorInPID1 rises. This flags are used by the sendOutputGain process to determine the output.

PIDTMR's purge logic removes the first PIDSimplex to fail. It's output logic uses the value of the output from PIDSimplex 0 and, after the first one fails, performs the logical AND operation between the other two. This approach generates an error when another internal PID controller fails.

PIDTMRSimplex's purge logic removes the first PIDSimplex to fail together with a working one. It's output logic uses the value of the output from PIDSimplex 0 and only changes if this specific internal PID controller is the first to fail. This causes the output to be the value of PIDSimplex 1 instead of PIDSimplex 0.

The testbench tb_PID_Matlab was used to validate the fault-tolerant PID controllers in a fault-free scenario. The result of the simulations was the same as shown in figure 4.4 as expected. A simulation with faults is required, but due lack of time a non-generic version of the fault injection framework proposed by the author in section 3.2 was developed.

The output buffers of PIDSimplex are the locations where the faults must be inserted. A FaultyBuffer class is derived from sc_buffer that enables injection. The overloaded write method forces a fault check to select between normal or faulty output, which is defined by the injected fault.

The method setFault is used at constructor of the module that contains a FaultyBuffer. The time-to-inject of the fault is defined by a random number, defined in an exponential distribution native of C++11. As the case of study is working with simulation time, λ is equal 1 for simplicity. The random number is added with 5, because the input step of the model rises at 5 seconds and is desired to consider that faults can occur only when the system "starts" (has an input different from zero). The result multiplied by 1000 is the injection time, moment in milliseconds of simulation time when a fault is injected. This operations result in the distribution presented by the equation 4.2. The value generated is then used to create an instance of BufferFault and register it at the FaultDatabase.

$$f(t) = (5 + e^{-t}) * 1000 \tag{4.2}$$

BufferFault encapsulates information for fault injection at FaultyBuffer, like the location of injection, the pointer of the faulty output method and the time-to-inject. FaultDatabase creates a list of faults during build phase. It uses the singleton design pattern, so any class can access it during simulation phase for inspection or manipulation of registered faults.

The FaultManager activates before each iteration of the Control_System. It retrieves from the FaultDatabase the list of faults and check which are pending (time-to-inject is equal or less the current simulation time). Those are injected in the respective FaultyBuffers and removed from the list.

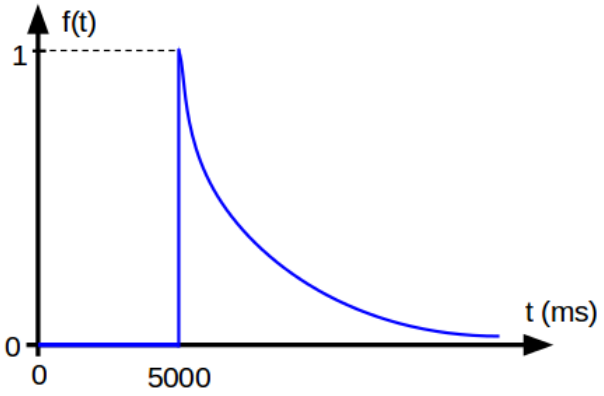

Figure 4.10: Distribution of time-to-inject of FaultyBuffer.

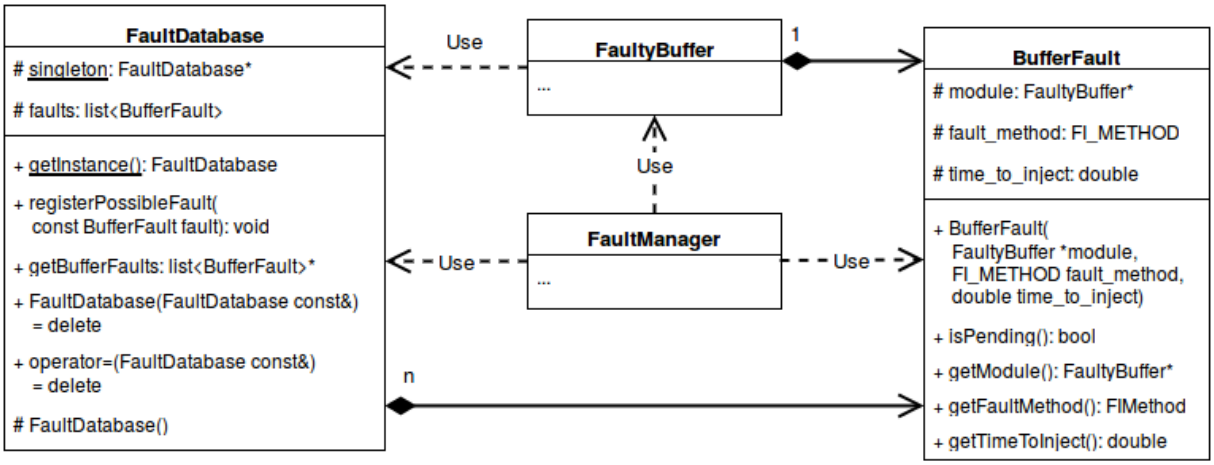

Figure 4.11: UML diagram of FaultModel, with FaultDatabase and BufferFault in details.

The testbench tb_PID_Matlab was modified to include the fault injection framework. The PIDSimplex buffers were changed by FaultyBuffers, FaultManager was instantiated at the class tb_PID_Matlab and a flag enable_faults enables the injection of faults. Simulations with the three implementations of PID controllers present expected behaviours, as shown at figure 4.14.

Fault tolerance techniques are used to increase the reliability of a system. The implemented PID controllers are validated checking their reliability according to the discussion in section 3.2. The output of PID controller was compared with the expected values and, when a discrepancy happens, the time-to-fail is registered and the simulations ends. The script run_N_simulations.tcl runs a configurable amount of simulations. A total of 10000 simulations generated the data of each curve in plot of figure 4.15.

The reliabilities of the implementations are similar to the expected curves in figure 3.5, validating the techniques' reliability. However, the curves won't be identical once the

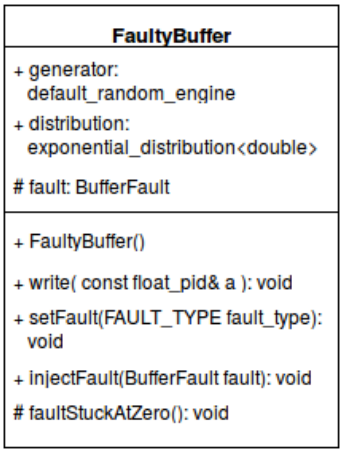

Figure 4.12: UML diagram of FaultyBuffer

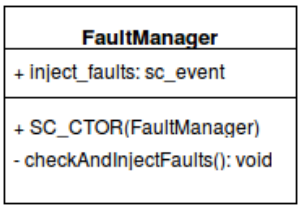

Figure 4.13: UML diagram of FaultManager

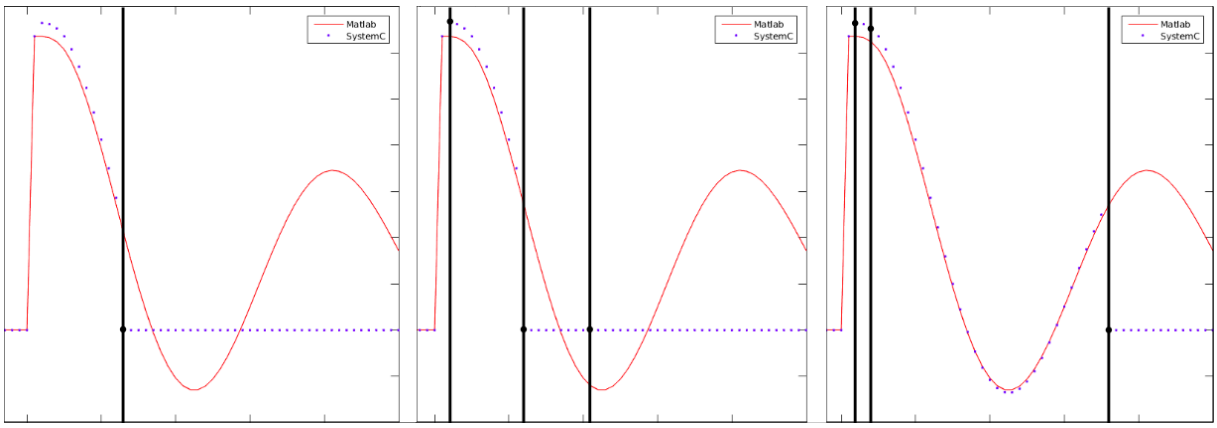

Figure 4.14: Simulation with faults of PIDSimplex, PIDTMR and PIDTMRSimplex, respectively. The black vertical bars represent a fault injection.

simulation pace is of 0.1 second and this causes all the times-to-inject to be rounded. This operation changes the curve and even creates discontinuities, like the horizontal lines in figure 4.15 from 5 to 5.1 seconds and 7.3 to 7.4 seconds, for example.

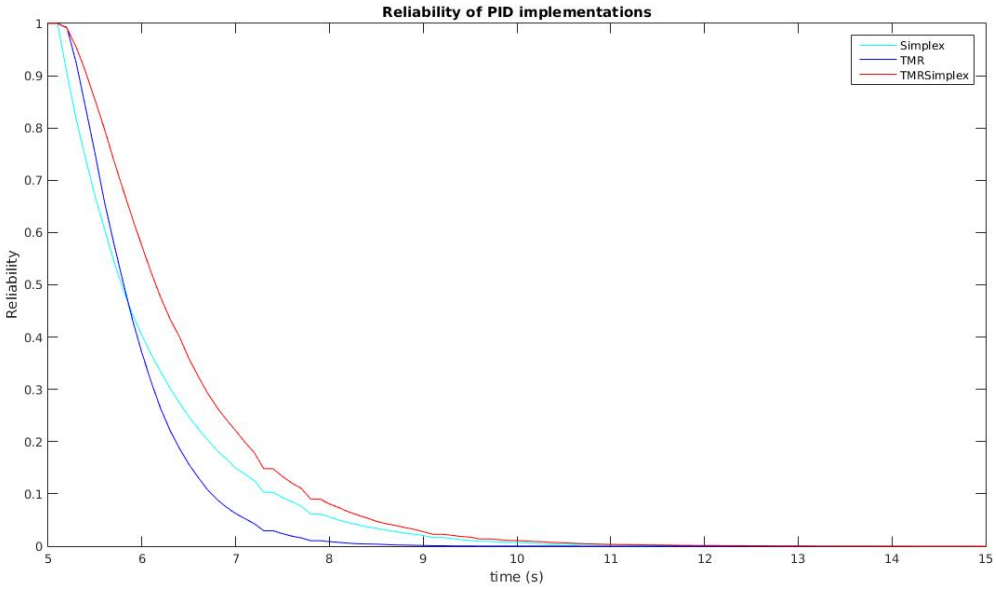

Figure 4.15: Reliability of each implementation of PID controller.

Chapter 5

Conclusion

This work's goal was to create a methodology of modeling a fault-tolerant control system, in a way that reliability can be measured and fault tolerance can be validated. Secondary goals were to the implement a case of study and to develop a fault injection framework. All of the objectives were met.

The methodology explained in chapter 3, result of this work, suggests a set of steps to create a validated fault-tolerant model. The specification of the model requires the type(s) of controller(s) to be used, transfer function in state-space notation of the plant and intended fault tolerance technique(s). An abstract class of the controller is created as the base of all child controllers, each one implementing a different fault tolerance technique. After unit test, the other control system modules are implemented, integrated and tested. A fault injection framework is applied to the model to validate the fault-tolerant controllers and check reliability of the system.

This methodology was used in the case of study of chapter 4. The testbench architecture and fault injection framework suggested in chapter 3 were used. The Matlab implementation presented not expected behaviour, possibly caused by wrong implementation of the model or configuration of the simulation. A more efficient verification technique is required to upgrade the methodology.

The architecture focused in modularization was very efficient for debug and design changes. Testbenches required minor changes in Stimulus, Monitor and top level classes to validate multiple modules. Different types of implemented PIDs could be replaced changing only a single line of code. It's a simple approach, but requires the recompilation of some or all files, which consumes a significant amount of time compared to simulation. A more simulation command-based configuration mechanism would be more interesting to remove the necessity of rerunning compilation.

The fault injection framework proposed worked well for the case of study, even though it was created a non-general version of it. To apply it, the ports of the controller are

exchanged with their faulty versions and possible faults are set in the constructor of the module. The fault tolerance techniques implemented present reliabilities similar to the expected when figures 3.5 and 4.15 are compared. However, the simulation time efficiency impact of it is unknown. This point is not in the scope of this work, but a significant delay must be inserted when compared to the fault-free simulation. Simple systems like the one in the case of study present no relevant time differences.

5.1 Future Works

A more general version of the fault injection framework can be implemented to enable reuse of it in other projects. As discussed above, it has a significant maintainability and the generalization is possible. The efficiency impact of it must be studied and compared to the cited authors' proposes.

The base created in this work can be used to study other fault tolerance techniques and its comparison with other implementations. Other approaches can be studied, implemented and tested to check reliability. Adaptability and reconfigurability are interesting alternatives to make a system tolerant to faults.

This work focused in permanent faults. Transient faults are a strong concerns with the small technologies used in microelectronics this days, as they can be generated by radiation for example. The adaptation of the fault injection model for insertion of transient and intermittent faults is an interesting step for further tolerance techniques analysis.

The Universal Verification Methodology (UVM), an industrial standard with great acceptance in this decade, can generate a testbench for VHDL or Verilog implementations of the system using a SystemC model. The library UVMConnect is required for that. This work's project can be used as a golden reference model after the correction of PID and control system outputs.

Referências

[1] Dubrova, Elena: *Fault-Tolerant Design*. Springer, KTH Royal Institute of Technology, Krista, Sweden, 2013. 1, 8, 9, 14, 15

[2] Hitt, Ellis F. and Dennis Mulcare: *The Fault-Tolerant Avionics*. CRC Press, Williamsburg, Virginia, 2001. 1

[3] Stengel, Robert: *Aircraft flight dynamics*, 2014. `https://www.princeton.edu/~stengel/MAE331Lecture10.pdf`, accessed in 2016. 2

[4] Nise, Norman S.: *Control Systems Engineering*. Wiley, California State Polytechnic University, Pomona, sixth edition, 2011. 3, 4

[5] Bricaud, Pierre and Michael Keating: *Reuse Methodology Manual for System-on-a-Chip Designs*. Kluwer Academic Publishers, New York, Boston, Dordrecht, London, Moscow, third edition, 2002. 5, 6

[6] Lee, Edward A and Alberto Sangiovanni-Vincentelli: *A framework for comparing models of computation*. IEEE Transactions on computer-aided design of integrated circuits and systems, 17(12):1217–1229, 1998. 5, 6, 7

[7] Grötker, Thorsten, Stan Liao, Grant Martin, and Stuart Swan: *System Design with SystemC*. Kluwer Academic Publishers, Hingham, Massachusetts, United States of America, second edition, 2002. 6, 7, 8

[8] Accellera: *About us and community of systemc*. `http://accellera.org/`, website acessed in 2016. 6

[9] Object Management Group : *Unified modeling language*. `http://www.omg.org/spec/UML/2.5`, accessed in 2016. 8

[10] Chang, Kun Jun and Yung Yuan Chen: *System-level fault injection in systemc design plataform*. In *2007 Proc. 8th Int. Symposium on Advanced Intelligent Systems*, pages 354–359, Chung-Hua University, Hsin-Chu, Taiwan, 2007. Citeseer. 9, 10, 16

[11] Shafik, Rishad A., Paul Rosinger, and Bashir M. Al-Hashimi: *Systemc-based minimum intrusive fault injection technique with improved fault representation*. pages 99–104, 2008. 9, 10, 16

[12] Black, David C., Jack Donovan, Bill Bunton, and Anna Keist: *SystemC: From the Ground Up*. Springer US, Texas, United States of America, second edition, 2010. 12

[13] Heesch, Dimitri van: *Doxygen*. www.doxygen.org/, accessed in 2016. 12

[14] Beauregard, Brett: *Improving the beginners pid*, 2011. http://brettbeauregard.com/blog/2011/04/improving-the-beginner%e2%80%99s-pid-derivative-kick/, website accessed in 2016. 12

[15] UNIVERSITY OF MICHIGAN, CARNEGIE MELLON UNIVERSITY, UNIVERSITY OF DETROIT MERCY: *Control tutorials for matlab and simulink: Aircraft pitch, system modeling*. http://ctms.engin.umich.edu/CTMS/index.php?example=AircraftPitch&section=SystemModeling, website accessed in 2016. 17, 18